DOCUMENT RESUME

ED 057 580 EM 009 420

AUTHOR Grant, Richard; And Others

TITLE LOGO Teaching Sequences on Numbers and Functions and Equations. Teacher's Text and Problems.
INSTITUTION Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY National Science Foundation, Washington, D.C.
REPORT NO R-2165
PUB DATE 30 Jun 71
NOTE 230p.; Programming-Languages as a Conceptual Framework for Teaching Mathematics, Volume Two; See also EM 009 419, EM 009 421, EM 009 422

EDRS PRICE MF-$0.65 HC-$9.87
DESCRIPTORS *Computer Assisted Instruction; *Mathematics Instruction; Numbers; *Programing Languages; Set Theory; *Teaching Guides
IDENTIFIERS Project LOGO

ABSTRACT

The teacher's texts for two teaching sequences in the LOGO mathematics course are presented in this second volume of a four-volume report. The material presented here is designed to be a broad overview of the application of LOGO to the topics of numbers and functions. A variety of alternative paths and approaches are presented; in each case the emphasis is on crucial points and on possible pitfalls and difficulties. The sequence on numbers is not meant to accompany a first exposure to the subject, but rather, a careful retracing of steps. The level of presentation in this unit is extremely detailed, and the reader is encouraged, on first reading, to skip around as his interests dictate. The sequence on functions is written more freely. The idea of function as a black-box is here concretely realized as are many other aspects of the set-theoretic approach to functions which otherwise trouble students by their "vagueness." For Volumes I, III, and IV of the report see EM 009 419, EM 009 421, and EM 009 422. (Author/JY)

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2165 Volume 2

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

LOGO Teaching Sequences on

Numbers

and

Functions and Equations

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

FOREWORD

The use of LOGO in the classroom requires that the teacher have a broad overview of the application of LOGO to the topics being treated. This volume contains materials intended to present such an overview in the topics of numbers and of functions. A variety of alternative paths and approaches are presented, in each case emphasis being placed on crucial points and on possible pitfalls and difficulties. The idea is not to present material in precisely the manner in which it is to be taught, but on giving the reader enough insight so that he can freely apply LOGO to the mathematical issues of direct concern to him. It may well be, for example, that only carefully selected portions are dealt with in the classroom, or that, a completely different approach is taken, arising perhaps from a suggestion in the text or the set of problems appended to each unit.

The material on numbers is not meant to accompany a first exposure to the subject, but rather, a careful retracing of steps. Not having to worry about what numbers are, the student can contrast the various sorts of number representations, concerning himself at each stage with understanding of the basic algorithms required. Writing these algorithms as LOGO procedures enables him to define them more precisely and concretely than otherwise, and gives him powerful means for using and extending them. The level of presentation in this unit is extremely detailed as befits the nature of the material and the reader is encouraged, on first reading, to skip around as his interests dictate.

The material on functions is much more suggestive and written more freely. The idea of function as black-box is here concretely realized as are many other aspects of the set-theoretic approach to functions which otherwise trouble students by their "vagueness".

Volume 2, Part 1

LOGO NUMBER UNIT

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

Richard Grant

Philip Faflick

Wallace Feurzeig

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

CONTENTS (continued)

LOGO Number Unit

Teacher's Text

1. Introduction to This Unit

The subject of this part of the LOGO mathematics course is numbers and their representation. It introduces several distinct ways of representing numbers, interprets the familiar operations of arithmetic in terms of each notational system, and develops the corresponding algorithms in the form of LOGO programs.

Four main representations are considered. We begin with a simple notation called mark numbers, similar to the marks used by children for scoring games. We extend this to a more compact notational system similar to Roman numerals, called compact numbers. We then introduce the idea of place value which makes possible a relatively pure form of positional notation, which we call place numbers. We evolve from this the relatively elaborate and efficient notation we know from familiar, everyday, current use as our numbers (these are called modern numbers, or mod numbers for short). Along the way we consider some related representations, such as "money numbers".

With each of these notations we develop LOGO procedures for performing the operations of arithmetic -- that is, for counting (succession), adding, comparing, subtracting, multiplying, and dividing -- with the corresponding numbers. We also write procedures for converting numbers across these various representations. Often several distinct algorithms for performing an operation (such as comparing mark numbers) are developed.

Throughout the text we have generally adopted the conventional use of "number" for what should properly be called "numeral".

In those instances where it is important to make this distinction, we use the term "representation" or "notation" (for example, as in "numbers expressed in this representation").

While this unit is a new presentation of number arithmetic, it is, at the same time, a unit on programming as a constructive means of expressing mathematical procedures, and this is an integral part of the presentation.

The text is written for use primarily by teachers rather than students, though the material is certainly appropriate for use by capable and advanced students as a basis for independent study and work. To help teachers in preparing their classroom discussions, we have included a great deal of simple expository material in the text.

The unit can be properly regarded as a re-introduction to arithmetic rather than as the first introduction. (We have distinctly different ideas about the first introduction of concepts like number and algorithm than those used here.) The material is intended for use at about seventh grade level though we think it would be equally suitable with many fifth grade classes. Like any sound presentation of good mathematics it is also appropriate, though in a somewhat different way for teaching older and more sophisticated students and teachers.

This unit illustrates one approach to the use of LOGO as a *conceptual framework* for teaching mathematics. It might appear feasible to teach the same material without using a computer, and indeed this is possible in principle. But, it would be

very difficult to do so in an effective way. An essential aspect of the presentation would be lost: we would no longer be providing the student with an active operational universe for constructing and controlling a mathematical process. This modified presentation would seriously impoverish the character and quality of the student's experience and of its educational benefit.

The main author of the unit is Richard Grant who advanced the central scheme, suggested the overall series of topics and their sequencing, and wrote the material on mark numbers and compact numbers. Philip Faflick wrote the chapters on place numbers and modern numbers. Wallace Feurzeig made occasional contributions to the ideas and writing, and edited the manuscript.

2. Mark Numbers

2.1 Introduction to Mark Numbers

Mark numbers are probably similar to the first way of expressing numbers man invented. We can imagine an ancient shepherd with a bag of pebbles, one pebble for every sheep. When he wants to count the sheep, he has them walk past in single file and removes one pebble from the bag as each sheep passes. If the bag is emptied just as the last sheep goes through, he knows that they are all there.

The bag of pebbles represents a number just as a word composed of X's does. They are both mark numbers; in the first pebbles are used as the marks, while in the second X's are used. Notches on the gun of a western gunfighter are another example.

An important virtue of mark numbers, besides their simplicity, is the ease with which one mark number can be changed into the next mark number. For example, by adding a pebble, an X, or a notch. This makes them very handy for keeping tallies; it never becomes necessary to erase the previous number in order to create the next. (The familiar modern form of mark numbers, which people often use in scoring games, for example 𝍸 𝍸 ||, has an added feature, grouping, that we shall discuss later.)

It is interesting to note that in writing a mark number we actually count up to it. For example, in order to write the mark number XXXXX, we must first write X, then writing another X we have XX, and then XXX, and so on.

2.2 Addition of Mark Numbers

There are at least two ways to add mark numbers. One way, just putting the two numbers together with the WORD operation, is exceptionally simple.

```
←TO ADD /M/ AND /N/
>1Ø OUTPUT WORD OF /M/ AND /N/
>END
←
```

With this algorithm, the two numbers are brought together into one. For example, with the bag of pebbles representation, adding two bags of pebbles is just pouring the pebbles into one bag. With mark numbers, adding is just putting all the X's from two words into one word. The first demonstration program, DEMO-ADDA, (see appendix for all demonstration programs) illustrates the two words coming together into one.

If we use the "notches on the gun" representation, the above method doesn't work very well. There is no way to take the notches on two guns and somehow move them on to a single gun. What we can do however is take a new gun and make notches on it, at the same time checking off the notches on one of the old guns. When all of these are checked, we continue with the other old gun. At the end, the new gun will have the sum of the notches from the two old guns. The idea here is that to add /N/ and /M/ we first count to /N/ and then continue counting /M/ more steps. The second demonstration program, DEMO-ADDB, illustrates this method of adding.

This method can be made more efficient by eliminating the first counting up to /N/. Instead, we can just start there and count /M/ more. For example, if we were given two sheets of paper,

one with the mark number for 1000 and the other with the mark number 37 and were told to hand in the sum, we could take a clean sheet and count to 1000 and then 37 more. On the other hand, we could take the sheet that already has 1000 written on it and just count 37 more.

2.3 Comparing Mark Numbers

It's reasonable to think, as we mentioned before, that numbers were invented for some purpose like making sure that the number of sheep that returned from pasture was the same as the number that set out. In other words, two numbers, the number that left and the number that returned, had to be compared to see if they were the same. If we want to compare two mark numbers on the computer, we can use the built-in operation IS. IS /M/ /N/ will output "TRUE" if the numbers are the same and "FALSE" if they are different. It has one drawback compared to any scheme the shepherd might have used to compare his two numbers. It doesn't tell which of the two is bigger! In fact, it's difficult to think of a method the shepherd could have used to find out whether two numbers were the same that wouldn't also tell which one is actually bigger.

Let's look at some algorithms for comparing mark numbers and their programs.

(1) Comparing the numbers /M/ and /N/ will give the same answer as comparing one less than /M/ with one less than /N/. But if we keep reducing the problem this way, eventually either /M/ or /N/ will become /EMPTY/. Then we'll know the other is bigger.

```
←TO COMPARE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/              (Is /M/ empty?)
>2Ø IF TRUE OUTPUT "SECOND" (If so, the second number /N/ is bigger)
>3Ø TEST EMPTYP OF /N/              (Similarly, is /N/ empty?)
>4Ø IF TRUE OUTPUT "FIRST"  (If so, the first number /M/ is bigger)
>5Ø OUTPUT COMPARE OF
     (BUTFIRST OF /M/) AND
     (BUTFIRST OF /N/)       (Get the answer for the reduced
>END                          problem.)
←
```

(2) We can start counting at /EMPTY/ (the zero mark number) and keep on until we get to one of the two numbers that we are comparing. The first one we reach will be the smaller number.

```
←TO COMPARE /M/ AND /N/
>1Ø MAKE                          (Start counting from /EMPTY/)
      NAME:  "COUNT"
      THING:  /EMPTY/
>2Ø TEST IS /COUNT/ /M/
>3Ø IF TRUE OUTPUT "SECOND"  (If /COUNT/ reaches /M/, /N/ is
                               bigger)
>4Ø TEST IS /COUNT/ /N/      (If /COUNT/ reaches /N/, /M/ is
>5Ø IF TRUE OUTPUT "FIRST"     bigger)
>6Ø MAKE                     (Otherwise, /COUNT/ is not big
      NAME:  "COUNT"          enough yet, so add one to it)
      THING:  WORD OF /COUNT/ AND "X"
>7Ø GO TO LINE 2Ø                        (Try again)
>END
←
```

(3) We can count backwards from the first number and see if we get to the second. If we do, the first number must have been the larger one.

```
←TO COMPARE /M/ AND /N/
>1Ø MAKE                          (Start backing up on /M/)
      NAME"  "M"
      THING:  BUTFIRST OF /M/
>2Ø TEST IS /M/ /N/
>3Ø IF TRUE OUTPUT "FIRST"        (We've come down to /N/ so /M/ must
                                   have started out larger)
>4Ø TEST EMPTYP OF /M/
>5Ø IF TRUE OUTPUT "SECOND"       (We've counted all the way down to
                                   /EMPTY/ without passing /N/, so
                                   /N/ must have been larger)
>6Ø GO TO LINE 1Ø                 (Try again)
>END
←
```

(4) We can type out the two numbers by typing first an X from the first number, then an X from the second number, and repeating this until one of the numbers is completely typed. The one that is finished first is the smaller number. The procedure DEMO-COMPARE uses this scheme.

```
←DEMO-COMPARE "XXX" AND "XXXXX"
FIRST     SECOND
  X         X
  X         X
  X         X
            X          (Now the program knows which number is
            X           larger but it finishes typing the larger
                        one anyway)
SECOND                 (... and announces the answer)
←
```

These are just some of the ways mark numbers can be compared. Possibly other ways will occur to students. One interesting way that works for people, but not for the computer, is to simply *look at* the two numbers together. The problem is that the operation of "looking at" is complex, for people as well as computers. But people have a working program for this, and the computer does not.

Even people have problems looking at some numbers. In fact, if we consider very large numbers, a person can't compare them at a glance and he would be forced to resort to some method similar to the programs we've just written. This is exactly what the prehistoric shepherd had to do to compare the number represented by the sheep with the number represented by the pebbles.

One small bug in all of the COMPARE procedures above is their behavior when the two inputs are the same. They will still output either "FIRST" or "SECOND". This can be easily fixed by adding a special check for equality and outputting, say, "EQUAL" if it is true. The first procedure might then look like this.

```
←TO COMPARE /M/ AND /N/
>5 TEST IS /M/ /N/
>6 IF TRUE OUTPUT "SAME"
>1Ø TEST EMPTYP /M/              (The rest is the same as before)
>2Ø IF TRUE OUTPUT "SECOND"
>3Ø TEST EMPTYP /N/
>4Ø IF TRUE OUTPUT "FIRST"
>5Ø OUTPUT COMPARE OF BUTFIRST OF /M/
    AND BUTFIRST OF /N/
>END
←
```

2.4 Subtraction of Mark Numbers

The difference between two numbers can be thought of in several ways. With mark numbers the most obvious way is to think of the difference between two numbers as the amount one number is bigger than the other. If we have two mark numbers,

XXXXXXXX and
XXXXX

then we can see that the difference is the number circled

XXXXXXXX
XXXXX

That is XXX. We can alter the first compare program so that

instead of outputting "FIRST" or "SECOND" it outputs the difference between the numbers. We'll use the fact that the difference between two numbers is the same as the difference between the two numbers both reduced by one. That is, the difference between XXX and XXXXX is the same as the difference between XX and XXXX which is the same as the difference between X and XXX which is the same as the difference between /EMPTY/ and XX. But, when one of the numbers is /EMPTY/, the difference is just the other number. So a program can be written

```
←TO DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/
>2Ø IF TRUE OUTPUT /N/          (If /M/ is /EMPTY/, the difference
                                 between /M/ and /N/ is all of /N/)
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT /M/          (Similarly in the opposite case)
>5Ø OUTPUT DIFFERENCE OF        (Neither is /EMPTY/ so the answer
    (BUTFIRST OF /M/) AND        is the same as the difference of
    (BUTFIRST OF /N/)            one less than each number)
>END
←
```

Notice that this program is essentially the same as the very first COMPARE program, except that different things are output in lines 20 and 40. The COMPARE program actually finds how big the difference between the numbers is, but it does not output that; it merely outputs either "FIRST" or "SECOND" to indicate which of the numbers is bigger.

An important characteristic of the difference between two numbers is that if we add this to the smaller number we get the larger one. We can use this fact to write a different kind of difference program. We'll call this one SUBTRACT. SUBTRACT has two inputs. It starts from /EMPTY/ and counts until it finds a number which, when added to the smaller input, gives the larger one.

```
←TO SUBTRACT /LARGER/ AND /SMALLER/
>1Ø MAKE
      NAME:  "COUNTER"
      THING:  /EMPTY/                    (Start the count at /EMPTY/)
>2Ø TEST IS /LARGER/ (ADD OF /COUNTER/
    AND /SMALLER/)
>3Ø IF TRUE OUTPUT /COUNTER/             (/COUNTER/ is the difference
                                          if /LARGER/ is the sum of
                                          /COUNTER/ and /SMALLER/)
>4Ø MAKE
      NAME:  "COUNTER"
      THING:  WORD OF /COUNTER/ AND "X"  (/COUNTER/ wasn't the
                                          difference so now
                                          increase it by one ...)
>5Ø GO TO LINE 2Ø                        (... and try again)
>END
←
```

This program is a little different from DIFFERENCE. With DIFFERENCE we could write the inputs in either order.

```
←PRINT DIFFERENCE OF "XXXX" AND "XX"
XX
←PRINT DIFFERENCE OF "XX" AND "XXXX"
XX
←
```

But, with SUBTRACT the larger number must come first. If we were to ask the computer to PRINT SUBTRACT OF "XX" AND "XXXX", it would try to find a number which it could add to "XXXX" to get "XX". It will check /EMPTY/, then X, then XX, then XXX, and so on. It will never find a number that works and so it will continue searching until someone interrupts it.

We could have avoided this "bug" by adding the lines

```
2 TEST IS COMPARE OF /LARGER/ AND /SMALLER/ "SECOND"
3 IF TRUE OUTPUT SUBTRACT OF /SMALLER/ AND /LARGER/
```

These lines cause the two inputs to exchange if /SMALLER/ is larger than /LARGER/.

Discussion about this "deficiency" of the natural numbers (or counting numbers or positive integers), that there is no number which can be added to "XXXX" so as to give "XX", will be used later in motivating the construction of negative numbers.

3. Compact Numbers

3.1 Grouping Mark Numbers

When mark numbers are used in scorekeeping these days, the most common form is 𝍸 𝍸 ||| where the marks are grouped into bunches of five marks each. A possible origin of this technique is finger counting. If we count on our fingers up to a large number, we'd be apt to think of the number as so many hands and so many fingers left over. Suppose a prehistoric general wanted to count the soldiers in his army. He could get an aide to help him. The aide would stand by and raise one finger as each soldier walked by him. After ten soldiers passed, all of the aide's fingers would be raised and the general would call another aide. This aide would count the next ten men on his fingers, and then the general would call still another aide. When the counting is over, the number of soldiers is represented by the aides used to determine it -- Marcus, Attila, Xenophon, Jura, Albert, and part of Hector (his left hand and thumb). This is much harder to remember than "fifty six" but it is much easier to remember than "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX".

We can write a procedure GROUP-COUNT that will count using mark numbers, but leave a space after every ten marks. Then, each group of marks stands for one man's fingers. One way to write this program is to use a subprocedure that types out ten marks. Then the main procedure will only have to call this subprocedure,

then type a space, and then go back to the beginning and start over. If this were all we wanted to do, then we could just use TYPE "XXXXXXXXXX" as the subprocedure and the program would work. (Remember TYPE is like PRINT except that it doesn't carriage return.)

```
←TO GROUP-COUNT
>1Ø TYPE "XXXXXXXXXX"
>2Ø TYPE /BLANK/                (Types a blank space)
>3Ø GO TO LINE 1Ø
>END
←

←GROUP-COUNT
XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX   (and so on)
```

This would be a more reasonable program if it actually counted something instead of just typing X's at high speed. One idea is to pause before typing a mark and wait until the ENTER key on the teletype is pressed, then typing one mark and waiting again. This way the computer will be counting the number of times the ENTER key is pressed.

The way to make the computer wait until ENTER is pressed is to use the REQUEST operation. For example,

```
←TO COUNT-ONE
>1Ø REQUEST        (The computer waits until the ENTER or RETURN
                    key is pressed)
>2Ø TYPE "X"       (... and then types "X")
>3Ø GO TO LINE 1Ø (... and then goes back to the beginning)
>END
←
```

This program counts in ordinary, ungrouped mark numbers. If we modify it so that it does the two commands REQUEST and TYPE "X" exactly ten times and then stops, we can use it in place of line 1Ø in GROUP-COUNT. It is easy to extend COUNT-ONE in this way.

All we need to do is write

```
←TO COUNT-TEN
>1Ø REQUEST
>2Ø TYPE "X"
>3Ø REQUEST
>4Ø TYPE "X"
>5Ø REQUEST
>6Ø TYPE "X"
>7Ø REQUEST
>8Ø TYPE "X"
>9Ø REQUEST
>1ØØ TYPE "X"
>11Ø REQUEST
>12Ø TYPE "X"
>13Ø REQUEST
>14Ø TYPE "X"
>15Ø REQUEST
>16Ø TYPE "X"
>17Ø REQUEST
>18Ø TYPE "X"
>19Ø REQUEST
>2ØØ TYPE "X"
>END
←
```

Now if we try COUNT-TEN it will count and write ten marks, so we can put it into GROUP-COUNT, as follows.

```
←TO GROUP-COUNT
>1Ø COUNT-TEN
>2Ø PRINT /BLANK/
>3Ø GO TO LINE 1Ø
>END
←
```

COUNT-TEN can be improved from a programming point of view just as SINGTWO was improved to become SINGALOT in the introductory unit. We can write COUNT-TEN so that it takes an input (a mark number) and types the number of marks of the input. If the input is /EMPTY/, it will stop. Otherwise, it will type one X and then shorten the input by one. Then it will start again with this shorter input.

```
←TO COUNTALOT /INPUT/
>1Ø TEST EMPTYP OF /INPUT/    (Are there any marks left in /INPUT/?)
>2Ø IF TRUE STOP              (if not, STOP)
>3Ø REQUEST
>4Ø TYPE "X"                  (Otherwise, COUNT-ONE)
>5Ø MAKE                      (Shorten the input by one)
      NAME:  "INPUT"
      THING:  BUTFIRST OF /INPUT/
>6Ø GO TO LINE 1Ø             (and go back to the beginning)
>END
←
```

We can shorten COUNTALOT a little more by using recursion. To do this, we notice that after we've typed the first X the work we have left to do is exactly what COUNTALOT (BUTFIRST OF /INPUT/) does. That is, we want to type one fewer X's than before. So we can write,

```
←TO COUNTALOT /INPUT/
>1Ø TEST EMPTYP OF /INPUT/
>2Ø IF TRUE STOP
>3Ø REQUEST
>4Ø TYPE "X"
>5Ø COUNTALOT (BUTFIRST OF /INPUT/)  (We've typed one X.  Now
                                       we'll type the rest of them)
>END
←
```

Now that we've changed COUNTALOT we have to go back to GROUP-COUNT. COUNTALOT needs an input. That input is the number of marks in each group, which could be any number. We decided to use ten, to represent the number of fingers on a man. So we write,

```
←TO GROUP-COUNT
>1Ø COUNTALOT "XXXXXXXXXX"
>2Ø TYPE /BLANK/
>3Ø GO TO LINE 1Ø
>END
←
```

We can, if we like, use groups of some size other than ten. We would only need to change line 1Ø of GROUP-COUNT. We will use

ten mark groups because they lead nicely into base ten number systems. If we were actually adopting group notation, however, it would probably be handier to use groups of five. The reason for this smaller grouping is that a reader can see at a glance whether a group has one, two, three, four, or five marks in it, while it is not so easy to distinguish groups of eight, nine, and ten marks. For groups that large it is helpful, even for people, to use some kind of discrimination procedure.

3.2 Conversion of Mark Numbers to Grouped Numbers

Since it's easier to read a mark number when it is grouped, we'll consider the problem of writing a procedure that takes an ordinary mark number and converts it to a grouped number. For example

```
←PRINT GROUP OF "XXXXXXXXXXXXX"
XXXXXXXXXX XXX
←
```

One of the easiest ways we've found to write this procedure is by using two subprocedures, FIRSTTEN and BUTFIRSTTEN. As the names suggest, these procedures are like FIRST and BUTFIRST except that they deal in groups of ten letters at a time. FIRSTTEN of a word is the first ten letters of the word; BUTFIRST-TEN of a word is the rest of the word, that is, the whole word except for the first ten letters.

We'll write GROUP first, assuming that we already have FIRSTTEN and BUTFIRSTTEN, and then we'll write those two procedures. One advantage to doing the writing in this order is that we can check out the soundness of the superstructure before investing time in working out its parts in detail. Further, we may discover, while writing GROUP, some special things FIRSTTEN or BUTFIRSTTEN should

do to make GROUP easier, whereas writing the two subprocedures first is not likely to help us in writing GROUP.

The basic idea behind the procedure GROUP is recursive. We'll reduce the problem of grouping a large word to that of grouping a smaller word. We'll keep this up until the problem is reduced to grouping the /EMPTY/ word. (That's easy. The result is /EMPTY/.)

In order to reduce grouping a large word to grouping a smaller one, we'll take the first ten marks of the word and put them into one group and then group the BUTFIRSTTEN of the word. To attach these two parts together as a group number, we can use the operation SENTENCE. SENTENCE OF (FIRSTTEN OF /WORD/) AND (GROUP OF BUTFIRSTTEN OF /WORD/) should take the first ten marks of /WORD/ and put a space between them and GROUP of the rest of the word. So GROUP is

```
←TO GROUP /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/      (If we've reduced /WORD/ all the
                                 way to /EMPTY/, then output the
                                 answer /EMPTY/)
>3Ø OUTPUT SENTENCE OF          (Otherwise, to get the answer, take
    (FIRSTTEN OF /WORD/) AND     the first ten, and GROUP of all
    (GROUP OF BUTFIRSTTEN OF     the rest and make a sentence of
    /WORD/)                      them.)
>END
←
```

Now we can write BUTFIRSTTEN. All we need to do is peel off the first ten marks. BUTFIRST OF /N/ will peel off one mark (BUTFIRST OF "XXX" is "XX"). BUTFIRST OF BUTFIRST OF /N/ will peel off two marks since what it says is take the BUTFIRST of whatever BUTFIRST of /N/ is. And finally, if we use ten BUTFIRST's, we can peel away ten marks.

```
←TO BUTFIRSTTEN /N/
>1Ø OUTPUT BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF /N/
>END
←
```

In writing FIRSTTEN we can use an important fact. If a word has ten or more marks in it, FIRSTTEN of the word is "XXXXXXXXXX". We don't have to actually use the first ten letters of the word itself by peeling off the last ones, or by using WORD to put together the first letter and the second and the third and so forth. We *would* have to do this if we wanted FIRSTTEN to work on *arbitrary* words (like "ABCDEFGHIJKLMNOP" or "ANTIDISESTABLISHMENTARIANISM"). But, since we're working with mark numbers, all of the letters in our words are X's.

What should FIRSTTEN do if a word has *fewer than* ten marks? A reasonable action would be to output the whole word. Reasonableness isn't an appropriate criterion, however, if it does not lead to our goal. Will this action by FIRSTTEN cause GROUP to work properly? Before we check that, let's take a look again at BUTFIRSTTEN. When we wrote that procedure, we didn't consider whether inputs had fewer than ten marks. The way we've written it, BUTFIRSTTEN will output /EMPTY/ for an input of ten or fewer marks. (Because BUTFIRST of /EMPTY/ is /EMPTY/.) So in the case of BUTFIRSTTEN we made a decision about small mark numbers more or less accidentally. We may have to go back and change it if it doesn't work out properly in GROUP.

Let's see what happens when GROUP gets an input smaller than ten X's, say "XXXX". First, "XXXX" isn't /EMPTY/ so we'll output the sentence of FIRSTTEN OF "XXXX" and GROUP OF BUTFIRSTTEN OF "XXXX". Assuming we've tentatively made the decisions for FIRSTTEN and

BUTFIRSTTEN that we felt were reasonable, then FIRSTTEN OF "XXXX" is "XXXX" and BUTFIRSTTEN OF "XXXX" is /EMPTY/. So we're outputting sentence of "XXXX" and GROUP OF /EMPTY/. But GROUP OF /EMPTY/ is /EMPTY/ because of lines 1Ø and 2Ø in GROUP. So we're outputting sentence of "XXXX" and /EMPTY/. That is just "XXXX". This says that when the input is a word smaller than ten marks, if we write FIRSTTEN to output the whole word and BUTFIRSTTEN to output /EMPTY/, then GROUP of the input word will be the word itself. Thus, GROUP OF "XXXX" will be "XXXX". But this is exactly what GROUP is supposed to do in this case, and what happens here with "XXXX" is typical of what should happen to any mark number with ten or fewer marks.

Thus, we'll write FIRSTTEN so that it outputs the whole word if the input has fewer than ten marks. The procedure will test if there are at least ten marks. If there are, it will output "XXXXXXXXXX"; if there aren't, it will output the whole word. A way to test whether there are at least ten marks is to peel off nine, using nine BUTFIRST's and then see if there are any left.

```
←TO FIRSTTEN /INPUT/
>1Ø TEST EMPTYP OF BUTFIRST OF      (Is there anything left after
    BUTFIRST OF BUTFIRST OF          peeling off nine marks?)
    BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF
    /INPUT/
>2Ø IF FALSE OUTPUT "XXXXXXXXXX"  (If so, /INPUT/ must have had at
                                    least ten marks since it wasn't
                                    /EMPTY/ after nine were peeled
                                    away)
>3Ø IF TRUE OUTPUT /INPUT/        (If not, /INPUT/ has fewer than
                                    ten marks so output the whole
                                    input word)
>END
←
```

Now GROUP should work. Let's test it.

```
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXX
←
```

It worked with that input. That's a good sign but it's not completely convincing in showing that GROUP will work for all inputs. How can we make a test that will be more convincing? Of course, we can't simply test *every* possible input since there are an infinity of them. What we can do is to test a representation of each different type of input. That is, each input which the procedure seems to handle differently. In this case we can properly use:

"XXXXXX" An input with fewer than ten marks.
"XXXXXXXXXX" Exactly ten marks.
"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" An input that groups evenly with no marks left over.
"XXXXXXXXXXXXXXXXXXXXXX" Several groups and some left over.

If GROUP works with these four inputs, we can be fairly sure it will always work; not perfectly sure, since we might have overlooked an important type, but nearly so. This way of choosing test inputs is much more effective in finding bugs than choosing test inputs at random.

```
←PRINT GROUP OF "XXXXXX"
XXXXXX
←PRINT GROUP OF "XXXXXXXXXX"
XXXXXXXXXX
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXXXXXXXXX
←PRINT GROUP OF "XXXXXXXXXXXXXXXXXXXXXX"
XXXXXXXXXX XXXXXXXXXX XXX
←
```

So GROUP probably works. There is an additional feature in LOGO that will let us watch more closely and exactly now GROUP works. This feature, called TRACE, will cause the teletype to print the inputs of GROUP each time it is called. For example,

```
←TRACE GROUP                          (This turns on the trace feature)
←PRINT GROUP OF "XXXXXXXXXXXXXX"
GROUP OF "XXXXXXXXXXXXXX"             (GROUP is first called with this
                                       input)
  GROUP OF "XXXX"                     (Next, GROUP is called with BUTFIRST-
                                       TEN of that input.  This calling
                                       happens at line 3Ø in GROUP)
    GROUP OF ""                       (To compute GROUP OF "XXXX" the com-
                                       puter needs to output sentence of
                                       FIRSTTEN of "XXXX" and GROUP of
                                       BUTFIRSTTEN of "XXXX", that is GROUP
                                       of /EMPTY/, or as written, GROUP OF
                                       "")
    GROUP OUTPUTS ""                  (GROUP OF "" can be computed immedi-
                                       ately, line 2Ø, and doesn't call
                                       GROUP again)
  GROUP OUTPUTS "XXXX"                (Now that GROUP OF "" has been com-
                                       puted, GROUP OF "XXXX" can output)
GROUP OUTPUTS "XXXXXXXXXX XXXX"       (Finally, GROUP OF "XXXXXXXXXXXXXX"
                                       can output since GROUP OF "XXXX" is
                                       now known)
XXXXXXXXXX XXXX                       (PRINT types out the final result)
←ERASE TRACE GROUP                    (This turns off the trace feature)
←
```

This TRACE feature is actually most useful when a procedure doesn't work properly at first and we want to know what is wrong. We turn on the TRACE feature and run our procedure with an input that doesn't work. Then, by looking at the printout we can often see where things are going wrong.

Notice the way inputs and outputs pair in the TRACE printout.

```
←TRACE GROUP
←PRINT GROUP OF "XXXXXXXXXXXXXX"
┌GROUP OF "XXXXXXXXXXXXXX"
│ ┌GROUP OF "XXXX"
│ │ ┌GROUP OF ""
│ │ └GROUP OUTPUTS ""
│ └GROUP OUTPUTS "XXXX"
└GROUP OUTPUTS "XXXXXXXXXX XXXX"
 XXXXXXXXXX XXXX
←
```

If the outermost GROUP gave the wrong answer, we could check the others to see where the error started. In this way we can usually find a very simple case where the procedure goes wrong. Then, by pretending to be the computer, we can examine that case in great detail and find the bug.

3.3 A Compact Representation of Grouped Numbers

So now we can have mark numbers typed out in groups of ten. We can go a step further and instead of typing out ten X's in each group, we can use another letter to represent a whole group. For example, if we choose T to stand for ten marks, then the number XXXXXXXXXXXXXXXXXXXXXX, or XXXXXXXXXX XXXXXXXXXX XXX, would be written TTXXX; that is, two groups of ten marks and XXX left over. One procedure for doing this is very similar to the one used in writing GROUP. In fact, we will be able to use the same programs we used there, with some minor changes. First, wherever GROUP had a word of ten X's we now want a T. The words of ten X's always come from line 2Ø of FIRSTTEN. So we'll change that to output T instead of XXXXXXXXXX.

```
←TO FIRSTTEN /INPUT/
>1Ø TEST EMPTYP OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF BUTFIRST OF BUTFIRST OF
    BUTFIRST OF /INPUT/
>2Ø IF FALSE OUTPUT "T"      (Instead of the ten X's)
>3Ø IF TRUE OUTPUT /INPUT/   (/INPUT/ has fewer than ten marks so
                              the X's are output, as before,
                              instead of a T)
>END
←
```

Now we'll try GROUP with this changed subprocedure.

```
←PRINT GROUP "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
T T T XXXXXX
←
```

This is almost exactly what we want. The only things wrong with it are the spaces. These are due to the line in GROUP that says OUTPUT SENTENCE OF ... . If we change SENTENCE to WORD, the various parts will be put together without spaces. While we make this change let's also change the name of the procedure from GROUP to something else, since its function has changed. It now converts a mark number to a different kind of number than a grouped mark number. We'll use the name COMPACT for the procedure and call numbers made up of these T's and X's *compact numbers*.

```
←TO COMPACT /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF FIRSTTEN OF /WORD/ AND COMPACT OF
    BUTFIRSTTEN OF /WORD/
>END
←
```

COMPACT is exactly the same as GROUP except that WORD replaces sentence in line 3Ø. Let's try it out.

```
←PRINT COMPACT OF "XXX"
XXX
←PRINT COMPACT OF "XXXXXXXXXX"
T
←PRINT COMPACT OF "XXXXXXXXXXXXXXXXXXXXX"
TTXX
←PRINT COMPACT OF "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
TTT
←
```

So COMPACT seems to work.

Compact numbers have one big advantage over mark numbers. They are much smaller, that is, they have fewer marks. The mark number that corresponds to TTTTTTTXXX wouldn't fit on one line. Because of their size, compact numbers are easier to write and easier to read. On the other hand, counting, adding, subtracting, and comparing are somewhat harder with compact numbers. TX is a bigger number than XXXXXXXXX despite appearances. Also, we can no longer count just by adding a mark to the preceding number. Sometimes it works, but every tenth time we have to rewrite the number, changing the nine X's into one T.

The advantages above seem, in practice, to far outweigh the disadvantages. As far back as history goes men used some form of compact numbers, usually with more than two different marks (Roman numerals or Babylonian cuneiform for example). Roman numerals, in fact, have been going out of use only gradually over the last thousand years and are still used today when a second number system is needed. Examples are for numbering chapters in a book, pages in an introduction, or topics in an outline.

3.4 Conversion of Compact Numbers to Mark Numbers

We have a program, COMPACT, that converts mark numbers to compact numbers. We could also use a program that goes in the opposite

direction, converting compact numbers to mark numbers. This is useful because mark numbers are, in many ways, easier to deal with than compact numbers. They are easier to compare and to add, for example. (We'll see later that mark numbers are also easy to multiply and divide.)

A simple idea for writing this program is to go through the compact number, leaving the X's alone and changing each T to XXXXXXXXXX. What we can do is take a T away from the front of the compact number and add XXXXXXXXXX to the back. Then we can do this over and over again until there are no more T's at the front of the word. For example, TTX would be changed to TXXXXXXXXXXX by taking off the first T and adding on the ten X's. Then TXXXXXXXXXXX would become XXXXXXXXXXXXXXXXXXXX. This number doesn't begin with T, so we're done.

```
←TO UNCOMPACT /N/
>1Ø TEST IS FIRST OF /N/ "T"     (Does the number begin with T)
>2Ø IF FALSE OUTPUT /N/          (It doesn't, so all the T's must
                                  be gone.  So it is a mark number
                                  and we should output it)
>3Ø MAKE                         (Remove the first T)
      NAME:  "N"
      THING:  BUTFIRST OF /N/
>4Ø MAKE                         (And put ten X's on the end)
      NAME:  "N"
      THING:  WORD OF /N/ AND    (Repeat the process with the
              "XXXXXXXXXX"        partially converted number, taking
>5Ø GO TO LINE 1Ø                 care of the next T if there is one)
>END
←
```

Now we should test UNCOMPACT. XXXXX and TTXX and TTT seem to be representative cases.

```
←PRINT UNCOMPACT OF "XXXXX"
XXXXX
←PRINT UNCOMPACT OF "TTXX"
XXXXXXXXXXXXXXXXXXXXX
←PRINT UNCOMPACT OF "TTT"
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
←
```

An amusing way to test UNCOMPACT is to notice that COMPACT OF UNCOMPACT OF /X/ should be /X/ and UNCOMPACT OF COMPACT OF /X/ should also be /X/. (Provided, of course, that /X/ is a compact number in the first case and a mark number in the second.)

```
←PRINT COMPACT OF UNCOMPACT OF "TTTXX"
TTTXX
```

Since UNCOMPACT OF "TTTXX" is "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX" and COMPACT of that is "TTTXX".

```
←PRINT UNCOMPACT OF COMPACT OF "XXXXXXXXXXXXX"
XXXXXXXXXXXXX
```

Since COMPACT OF "XXXXXXXXXXXXX" is "TXXX" and UNCOMPACT OF "TXXX" is "XXXXXXXXXXXXX".

It is also interesting to try COMPACT OF UNCOMPACT and UNCOMPACT OF COMPACT on words that aren't numbers (for example, "ABCDEFGHIJKLMNOPQRSTUVWXYZ") to see what they do. A good problem is to describe exactly which words these two operations (COMPACT OF UNCOMPACT and UNCOMPACT OF COMPACT) leave unchanged. The effect of these operations might be surprising. For example,

```
←PRINT COMPACT OF UNCOMPACT OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
TTUVWXYZ
```

The explanation is straightforward. Since "ABCDEFGHIJKLMNOPQRST-UVWXYZ" doesn't begin with "T", UNCOMPACT leaves it unchanged. COMPACT substitutes a "T" for the first ten marks, "ABCDEFGHIJ", and another for the next ten, "KLMNOPQRST", and leaves the last six unchanged.

3.5 Disordered Compact Numbers

So far all the compact numbers we've looked at have had all T's to the left of the X's. But, what about a number like XXXTTX? If we follow the rule that a T stands for ten X's, then that number is the same as XXXXXXXXXXXXXXXXXXXXXXX. That is, it is the same as TTXXXX or TXXXXT or several other forms, such as XXXXXXXTXXXXXXX. There are many ways of writing this number, and the order of marks really doesn't matter at all. It doesn't even matter if the number isn't completely compacted. It is always clear which number is meant. (We're ignoring the commonly used subtractive notation of Roman numerals which makes IV the same as IIII. The only important modern use of this is in telling time anyway.)

We shall write an uncompacting procedure to work for any form of compact number. Notice that our old UNCOMPACT doesn't work with most compact numbers. It will convert the T's on the left end of the word but not any others.

```
←PRINT UNCOMPACT OF "XXTTT"
XXTTT
←PRINT UNCOMPACT OF "TXTXT"
XXXXXXXXXXTXT
```

The simplest way to write this procedure is to use recursion. We note that if a word begins with X, then UNCOMPACT OF the word is the same as UNCOMPACT OF BUTFIRST OF that word, with an X stuck on the end. If the word begins with T, then UNCOMPACT of the word is the same as UNCOMPACT OF BUTFIRST OF the word, with XXXXXXXXXX stuck on the end. So, UNCOMPACT works on the word by working on the BUTFIRST of the word. And UNCOMPACT works on the BUTFIRST of the word by working on the BUTFIRST OF BUTFIRST of the word. And so on. With each time around, the word gets smaller.

Eventually it will be /EMPTY/. And UNCOMPACT OF /EMPTY/ is /EMPTY/, so the process terminates.

```
←TO UNCOMPACT /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT /EMPTY/     (If /N/ is /EMPTY/, the answer is
>3Ø TEST IS FIRST OF /N/ "X"                          /EMPTY/)
>4Ø IF TRUE OUTPUT WORD OF (UNCOMPACT
    OF BUTFIRST OF /N/) AND "X"    (FIRST is "X", so tack "X"
                                    onto UNCOMPACT of BUTFIRST)
>5Ø OUTPUT WORD OF (UNCOMPACT OF
    BUTFIRST OF /N/) AND
    "XXXXXXXXXX"                (FIRST is not "X" so it must be T.
                                 Hence tack "XXXXXXXXXX" onto
                                 UNCOMPACT OF BUTFIRST)
>END
←
```

Let's look at this procedure. Obviously it works for empty inputs. If /N/ is /EMPTY/, then UNCOMPACT will output /EMPTY/, the correct response. What happens if /N/ is only one letter long? Well, if /N/ is "X", then line 4Ø is used and UNCOMPACT outputs WORD OF UNCOMPACT OF /EMPTY/ AND "X". But UNCOMPACT OF /EMPTY/ is /EMPTY/ as we just saw, so this is WORD OF /EMPTY/ AND "X" or "X". Thus, UNCOMPACT OF "X" is "X". If /N/ is "T", then line 5Ø is used and UNCOMPACT outputs WORD OF UNCOMPACT OF /EMPTY/ AND "XXXXXXXXXX". That is WORD OF /EMPTY/ AND "XXXXXXXXXX", or "XXXXXXXXXX". So UNCOMPACT OF "T" is "XXXXXXXXXX". UNCOMPACT works for both of the one-letter inputs. We could continue like this, showing that UNCOMPACT works for all two-letter inputs and for all three-letter inputs, and so on. Psychologically this can be very convincing. After a short time (say, when we've proved it for up to seven-letter inputs), anyone but a mathematician probably would be completely assured that UNCOMPACT will always work. The mathematician would ask how we could be sure that the first input for which UNCOMPACT fails isn't some huge word consisting of millions of letters.

Fortunately, there are ways to avoid this problem. One is called mathematical induction (or, in the form we will use, the method of infinite descent). We will show that if there is any compact number that UNCOMPACT doesn't work on, then there is a smaller compact number (BUTFIRST of the original number) on which UNCOMPACT also doesn't work. But, then we'll be able to apply this argument again to the new number and get a still smaller number on which UNCOMPACT won't work. We can continue this way until eventually we'll be down to a one-letter number which UNCOMPACT can't handle. But there are only two one-letter numbers, X and T, and UNCOMPACT works perfectly on both of them. So this method will show that there can't be any compact number that UNCOMPACT won't correctly convert into a mark number.

Well, all we need to show is that if UNCOMPACT doesn't work for some compact number, then it also doesn't work for the BUTFIRST of that number. The number begins with either X or T. If it begins with X, then BUTFIRST of it is a compact number one less than the original number. So UNCOMPACT OF BUTFIRST OF the original number should have one mark less than UNCOMPACT OF the original number. But, the way the procedure for uncompacting works in this case (line 4Ø of UNCOMPACT) is to tack one more X onto the UNCOMPACT OF BUTFIRST. So if UNCOMPACT of the number itself is wrong, then UNCOMPACT of the BUTFIRST of the number must have been wrong.

The same argument is true if the original number began with T. We'd refer to line 5Ø of the procedure and to tacking on XXXXXXXXXX instead of line 4Ø and tacking on X.

This argument will convince the mathematician that UNCOMPACT will work for any compact number. It is a curious fact that the earlier, incomplete argument is much more likely to be convincing

to nonmathematicians. This is probably because few nonmathematicians have any fluency with complex *logical* arguments. Arguments in life situations nearly always hinge on questions of evidence or values rather than subtle logical points.

The reason for choosing mathematical induction for the test of UNCOMPACT is the close relationship between induction and recursion. When we write a recursive procedure, we decide to solve the problem of a large input by solving the same problem for a smaller input. Eventually the input becomes small enough so that the solution is trivial (usually we let the input get down to /EMPTY/). With a related form of mathematical induction, we say that if the solution is wrong for a large input it must be wrong for a smaller input. Then it must be wrong for a still smaller input. Finally, the solution must be wrong for the case of /EMPTY/. But, we can check that case and see that is correct. So, the solution can't be wrong for any input.

Tracing will show how the UNCOMPACT procedure makes the problem simpler and simpler until it can finally solve it.

```
←TRACE UNCOMPACT
←PRINT UNCOMPACT OF "XTX"
UNCOMPACT OF "XTX"                (The original problem)
  UNCOMPACT OF "TX"               (We can solve the original problem
                                   if we know the answer to this one.
                                   The solution to the original prob-
                                   lem is WORD OF (UNCOMPACT OF "TX")
                                   AND "X")
    UNCOMPACT OF "X"              (Now we can solve the problem just
                                   above if we can solve this problem.
                                   The solution to the problem above,
                                   UNCOMPACT OF "TX" is WORD OF
                                   (UNCOMPACT OF "X") AND "XXXXXXXXXX",
                                   by line 5Ø of UNCOMPACT.  If we
                                   substitute this into the solution
                                   of the original problem above we get
                                   that, the solution is WORD OF (WORD
                                   OF UNCOMPACT OF "X" AND "XXXXXXXXXX")
                                   AND "X")
```

```
      UNCOMPACT OF ""              (BUTFIRST OF "X" is /EMPTY/, also
                                    written "".  By line 4Ø of UNCOMPACT
                                    we have UNCOMPACT OF "X" is WORD OF
                                    (UNCOMPACT OF "") AND "X")
      UNCOMPACT OUTPUTS ""         (The problem UNCOMPACT OF "" is al-
                                    ready as simple as possible.  The
                                    answer is "")
    UNCOMPACT OUTPUTS "X"          (Now that UNCOMPACT OF "" is solved
                                    UNCOMPACT OF "X" is known.  It was
                                    just WORD OF (UNCOMPACT OF "") AND
                                    "X")
  UNCOMPACT OUTPUTS "XXXXXXXXXXX"  (Remember that UNCOMPACT OF "TX"
                                    was WORD OF (UNCOMPACT OF "X") AND
                                    "XXXXXXXXXX".  Now that UNCOMPACT
                                    OF "X" is solved, UNCOMPACT OF "TX"
                                    is known)
UNCOMPACT OUTPUTS "XXXXXXXXXXXX"   (Finally the original problem
                                    is solved now that UNCOMPACT OF
                                    "TX" is known)
XXXXXXXXXXXX                       (And the result is printed)
←
```

Despite all these discussions about UNCOMPACT, we should still test it with some representative inputs just to make triply sure it works.

```
←PRINT UNCOMPACT OF "XT"
XXXXXXXXXXX
←PRINT UNCOMPACT OF "XXXXXX"
XXXXXX
←PRINT UNCOMPACT OF "TT"
XXXXXXXXXXXXXXXXXXX
←
```

We can write programs to do arithmetic with compact numbers by using UNCOMPACT to convert the compact numbers into mark numbers, using the old mark number procedures ADD, SUBTRACT, and so on, and then using COMPACT to convert the answer back into a compact number. This method would work but it is inefficient. It preserves the advantages of compact numbers for the people using the computer but not for the computer itself. It would be silly

to worry about being "fair" to the computer except that the more work we make the computer do the longer it takes. And long waits for an answer are annoying to people. So it is to our advantage to allow the computer to operate efficiently.

In handling compact numbers, it is a great convenience to be able to separate the T's and the X's. Suppose we have two procedures, T'S OF /N/ and X'S OF /N/ (the ' can be used in the name of a LOGO procedure just as if it were a letter) that work as follows:

```
←PRINT T'S OF "XXTTXT"
TTT
←PRINT X'S OF "XXTTXT"
XXX
←PRINT T'S OF "T"
T
←PRINT X'S OF "T"
      (/EMPTY/ is printed)
←
```

T'S outputs a word made up of all the T's in its input and X'S does the same for the X's. These two procedures can be used for straightening out disordered compact numbers, for comparing compact numbers (since we have to compare the T's and then, if both have the same number of T's, compare the X's), and so forth.

We'll write T'S and X'S. Let's do T'S first. Then X'S should turn out to be a trivial modification of T'S. We'll write T'S using recursion. All we need to do is to notice that if a word begins with X, then T'S OF that word is the same as T'S OF BUTFIRST OF the word. If the word begins with T, then T'S OF the word is just T'S OF BUTFIRST OF the word, with an extra T stuck on at the end. That is, WORD OF (T'S OF BUTFIRST OF the word) AND "T". So whether the word begins with T or X, we can reduce the problem to getting T'S OF BUTFIRST OF the word. We can keep doing this

until the problem is reduced to finding T'S OF /EMPTY/. This is a trivial question and we can immediately give the answer, /EMPTY/.

```
←TO T'S /N/
>1Ø TEST EMPTYP OF /N/          (Have we gotten down to the trivial
                                 case yet?)
>2Ø IF TRUE OUTPUT /EMPTY/      (If so, the answer is /EMPTY/; there
                                 are no T's in the empty word)
>3Ø TEST IS FIRST OF /N/ "T"
>4Ø IF TRUE OUTPUT WORD OF (T'S (If /N/ begins with T, then the
    OF BUTFIRST OF /N/) AND "T"  answer is T'S OF BUTFIRST OF /N/
                                 with one more T stuck on)
>5Ø OUTPUT T'S OF BUTFIRST OF   (Otherwise the first letter of /N/
    /N/                          isn't T so the answer is just T'S
                                 OF BUTFIRST OF /N/. We can just
                                 forget the first letter)
>END
←
```

And, of course, we'll test the procedure.

```
←PRINT T'S OF "XXX"
      (The empty word, since there are no T's in XXX)
←PRINT T'S OF "TT"
TT
←PRINT T'S OF "XTTXTX"
TTT
←
```

The procedure X'S can be written exactly like T'S. We need only replace "T" in lines 3Ø and 4Ø by "X".

```
←TO X'S /N/
>1Ø TEST EMPTYP OF /N/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø TEST IS FIRST OF /N/ "X"
>4Ø IF TRUE OUTPUT WORD OF (X'S OF BUTFIRST OF /N/) AND "X"
>5Ø OUTPUT X'S OF BUTFIRST OF /N/
>END
←
←PRINT X'S OF "TTXTX"
XX
```

3.6 Standard Form

Now that we have the procedures T'S and X'S, it is a simple matter to write some compact number manipulators. For example, while it is true that order has no effect on the value of a compact number, it is also true that it is easier to read compact numbers if they are written in some standard form (say, all T's followed by all X's). So, let's write a procedure to convert any compact number to standard form. All the procedure needs to do is put the T's at the beginning and the X's at the end.

```
←TO STANDARDIZE /NUMBER/
>1Ø OUTPUT WORD OF (T'S OF /NUMBER/)
    AND (X'S OF /NUMBER/)
>END
←
```

We can try STANDARDIZE on some of the disordered compact numbers we mentioned before.

```
←PRINT STANDARDIZE OF "XTXTXX"
TTXXXX
←PRINT STANDARDIZE OF "XXXTT"
TTXXX
←PRINT STANDARDIZE OF "XX"
XX
←PRINT STANDARDIZE OF "TTXX"
TTXX
←PRINT STANDARDIZE OF "XXXXXXTXXXXXX"
TXXXXXXXXXXXX
←
```

STANDARDIZE works, up to a point. It does seem to arrange the T's and X's properly but it doesn't do any compacting. This is perfectly reasonable, since we didn't instruct it to do so when we defined the procedure. We could leave STANDARDIZE as it is and truthfully say that we've written what we set out to write, a program that will arrange the T's and X's of a compact number

so that the T's are to the left of the X's. It is an advantage, however, if any two standard compact numbers that stand for the same mark number are exactly the same. (For example, if we wanted to see if two compact numbers were equal, we could just do IS OF STANDARDIZE OF one AND STANDARDIZE OF the other.) So, let's add to our criteria for standard form that there be no more than nine X's in the number. So, a compact number is in standard form only if (a) all the T's are on the left, and (b) there are no more than nine X's. The reader should convince himself that these rules guarantee that two equal compact numbers will be exactly the same.

In order to fix STANDARDIZE so that it will output numbers in our newly-defined standard form, we need to adjust the phrase (X'S OF /NUMBER/). The problem occurs when (X'S OF /NUMBER/) is a word of more than nine X's. COMPACT is a program that will handle this problem. COMPACT will take the word of X's and output a word with nine or fewer X's, the extra X's compacted into T's. As a special piece of fortune, COMPACT puts those extra T's to the left of the X's, just the right place to connect them properly with the T's output from (T'S OF /NUMBER/). So we can write

```
←TO STANDARDIZE /NUMBER/
>1Ø OUTPUT WORD OF (T'S OF /NUMBER/)
    AND (COMPACT OF X'S OF /NUMBER/)
>END
←
```

If we consider STANDARDIZE OF "XXXXXXTXXXXXX", we can see that the procedure will output WORD OF "T" AND "TXX" which is "TTXX".

```
←PRINT STANDARDIZE OF "XXXXXXTXXXXXX"
TTXX
```

And, of course, we should test the new STANDARDIZE on a representative group of compact numbers.

```
←PRINT STANDARDIZE OF "TT"
TT
←PRINT STANDARDIZE OF "XX"
XX
←PRINT STANDARDIZE OF "XXXTT"
TTXXX
←
```

Now that we have STANDARDIZE, ADD is a trivial task. If we didn't care whether or not the answer was in standard form, then the elementary operation WORD would serve as a working ADD procedure. Simply putting the two compact numbers together gives a new number with exactly enough T's and X's to represent the sum of the two original numbers. But, as long as we can get any representation for this sum, we might as well get the standard form representation by using STANDARDIZE.

```
←TO ADD /X/ AND /Y/
>1Ø OUTPUT STANDARDIZE OF WORD OF
    /X/ AND /Y/
>END
←
```

ADD is so simple it is hard to think of test cases for which it might be wrong. The subprocedures we wrote to help us with ADD, T'S, X'S, and STANDARDIZE, turn out to be so powerful that our original problem, ADD, has become a triviality. This is a common occurrence in mathematics. It also often happens that tools (subprocedures, subtheorems, techniques) invented to attack some problem end up having an importance of their own, quite apart from the problem they were invented for.

3.7 Comparing Compact Numbers

Before we write a procedure to compare two compact numbers, let's assume that from now on compact numbers will be written in standard form, unless stated otherwise. Thus, the inputs to compact number procedures will be in standard form and the procedure must output numbers in standard form. This requirement for the outputs is obviously desirable. The requirement that the inputs be in standard form will make our procedures less powerful (they won't handle nonstandard inputs) but simpler. If, for any reason, we decide later that we want the extra power, we can convert each nonstandard input into standard form simply by adding the instruction line

```
MAKE
  NAME:  "INPUT"
  THING:  STANDARDIZE OF /INPUT/
```

at the beginning of the procedure (with "INPUT" replaced by "X" or "M" or "FIRST" or whatever name the procedure uses for the input). These lines will put the inputs into standard form and then we won't have to worry about them again.

In comparing two compact numbers, the first thing to look at is the T's. If one of the numbers has more T's than the other, that one must be the larger. This is because, with the numbers in standard form the difference of the X's can't be more than nine; not enough to make up for an extra T. Only if the number of T's is the same for both do we need to look at the X's. In that case, of course, the larger number is simply the one with the more X's.

Comparing the number of T's (or X's) in the two numbers is just a matter of comparing the number of letters in two words. But, that is a problem we solved when we wrote the procedures to compare mark numbers. One of the programs was

```
←TO MARK-COMPARE /FIRST/ AND /SECOND/   (We changed the name to
                                         MARK-COMPARE so that we
                                         can use the name COMPARE
                                         for the compact number
                                         program)
>1Ø TEST IS /FIRST/ /SECOND/
>2Ø IF TRUE OUTPUT "EQUAL"
>3Ø TEST EMPTYP OF /FIRST/
>4Ø IF TRUE OUTPUT "SECOND"
>5Ø TEST EMPTYP OF /SECOND/
>6Ø IF TRUE OUTPUT "FIRST"
>7Ø OUTPUT MARK-COMPARE OF (BUTFIRST
    OF /FIRST/) AND (BUTFIRST OF
    /SECOND/)
>END
←
```

When we wrote MARK-COMPARE (we called it COMPARE then), we intended that it should work for words made up entirely of X's. But, there is nothing in the program that takes the least notice of what letters make up /FIRST/ and /SECOND/. So MARK-COMPARE will work with any two words, outputting the name of the word that has the larger number of letters.

To write COMPARE (for compact numbers), we'll use MARK-COMPARE to compare the T's. If these aren't EQUAL, we're done and the larger number is simply the one with the more T's. If the T's are EQUAL, then the answer to the problem is just MARK-COMPARE of the X's.

```
←TO COMPARE /FIRST/ AND /SECOND/   (/FIRST/ and /SECOND/ will
                                    always be compact numbers in
                                    standard form)
>1Ø TEST IS (MARK-COMPARE OF T'S   (Do both inputs have the same
    OF /FIRST/ AND T'S OF           number of T's?)
    /SECOND/) "EQUAL"
>2Ø IF FALSE OUTPUT MARK-COMPARE   (If they don't, the one with
    OF (T'S OF /FIRST/) AND         the more T's is the larger)
    (T'S OF /SECOND/)
```

(continued)

```
>3Ø OUTPUT MARK-COMPARE OF
    (X'S OF /FIRST/) AND (X'S
    OF /SECOND/)
```

(If they do have the same number of T's, then the answer is the one with the more X's. Notice that if the numbers have equal T's and equal X's, this line will output "EQUAL")

```
>END
←
```

We'll test COMPARE with some representative inputs.

```
←PRINT COMPARE OF "TTX" AND "XXXX"
FIRST
←PRINT COMPARE OF "TTXX" AND "TTXXX"
SECOND
←PRINT COMPARE OF "TXX" AND "TXX"
EQUAL
←
```

We were quite specific in insisting that COMPARE would be guaranteed to work only on numbers in standard form. In fact, COMPARE will work properly with many, but not all, nonstandard compact numbers. Disordering presents no problem because T'S and X'S handle that satisfactorily. Incomplete compacting (more than nine X's) is what can cause problems. (Find an example for which COMPARE gives the wrong answer.)

3.8 Subtraction of Compact Numbers

This COMPARE procedure cannot be converted into a SUBTRACT program as easily as in the mark number case. An attempt to do so would go like this. We'll make the difference between two compact numbers a new compact number whose T's equal the difference in the T's of the two original numbers and whose X's equal the difference of the X's. To do this we'll use MARK-DIFFERENCE on the T's and X's separately. Then DIFFERENCE would be -

```
←TO DIFFERENCE /F/ AND /S/
>1Ø MAKE
      NAME:  "T'S OF ANSWER"      (Find the number of T's in the
      THING:  MARK-DIFFERENCE OF   answer)
              (T'S OF /F/) AND
              (T'S OF /S/)
>2Ø MAKE
      NAME:  "X'S OF ANSWER"      (Find the number of X's in the
      THING:  MARK-DIFFERENCE OF   answer)
              (X'S OF /F/) AND
              (X'S OF /S/)
>3Ø OUTPUT WORD OF /T'S OF        (Put the T's and X's together
    ANSWER/ AND /X'S OF ANSWER/    with WORD and output)
>END
←
```

Recall the mark number subtraction procedure, MARK-DIFFERENCE:

```
←TO MARK-DIFFERENCE /M/ AND /N/
>1Ø TEST EMPTYP OF /M/
>2Ø IF TRUE OUTPUT /N/
>3Ø TEST EMPTYP OF /N/
>4Ø IF TRUE OUTPUT /M/
>5Ø OUTPUT DIFFERENCE OF (BUTFIRST
    OF /M/) AND (BUTFIRST OF /N/)
>END
←
```

If we try out DIFFERENCE, we will find that it doesn't work in all cases.

```
←PRINT DIFFERENCE OF "TTX" AND "TX"
T
←PRINT DIFFERENCE OF "TXX" AND "TTTXXXX"
TTXX
←PRINT DIFFERENCE OF "TTX" AND "TXXXX"
TXXX
←
```

Here is a bug. The correct answer is XXXXXXX, not TXXX. It's clear how the error occurred. The first input has two T's and the second has one so the difference in the T's is T. The first

input has one X and the second four, so the difference in the X's is XXX. So, DIFFERENCE output TXXX. The problem is that the smaller number has more X's than the larger number. How can we handle this?

A solution to our problem is to partially uncompact the larger number, changing one of its T's to ten X's. Then it will certainly have more X's than the smaller number (which can have no more than nine, being in standard form). When we try this on the problem that gave us the trouble, DIFFERENCE OF "TTX" AND "TXXXX" we change it to DIFFERENCE OF "TXXXXXXXXXX" AND "TXXXX". The procedure DIFFERENCE will have no difficulty with this problem, the difference of the T's is /EMPTY/ and the difference of the X's is XXXXXXX. So, the answer is XXXXXXX.

If we try to fix DIFFERENCE this way, we'll have a little trouble because we'll have to first find out which of the two input numbers is larger before we can decide whether we need to do the uncompacting. This isn't particularly difficult to do -- we already have the COMPARE procedure we need. And, since DIFFERENCE is becoming somewhat complicated, instead of repairing it, we'll write a new procedure, SUBTRACT, which uses it as a subprocedure. SUBTRACT will be like DIFFERENCE except that it will require that the first of the two inputs be larger than the second.

```
←TO SUBTRACT /BIG/ AND /SMALL/
>1Ø  TEST IS (COMPARE OF X'S OF /BIG/      (Which number has more X's?)
     AND X'S OF /SMALL/) "SECOND"
>2Ø  IF TRUE MAKE                          (/SMALL/ has more X's, so
       NAME:  "BIG"                         change one of the T's in
       THING:  WORD OF (BUTFIRST OF         /BIG/ to ten X's)
               T'S OF /BIG/) AND
              (WORD OF X'S OF /BIG/
              AND "XXXXXXXXXX")
>3Ø  OUTPUT DIFFERENCE OF /BIG/ AND        (We've made sure that this is
     /SMALL/                                a case that DIFFERENCE can
>END                                        handle so we use it)
←
```

There are a few questions that need to be settled before we can have confidence in SUBTRACT. For example: how can we be sure, in line 2Ø, that /BIG/ has a T at all? If /BIG/ has no T, then, of course, we can't uncompact it. What line 2Ø does, in that case, is replace /BIG/ by another number, ten larger. That would be certain to foul up our subtraction. Fortunately, line 2Ø is carried out only if /SMALL/ has more X's than /BIG/ (because of the IF TRUE condition following the test in line 1Ø) and, if that is so, /BIG/ must have at least one T in order to be bigger than /SMALL/.

It is also important, if we do perform line 2Ø, that /BIG/ originally have more T's than /SMALL/ has. Otherwise, after line 2Ø is carried out, /SMALL/ will have more T's than /BIG/ and so the subprocedure DIFFERENCE won't work. (For example, consider DIFFERENCE OF "TXXXXXXXXXXXXX" AND "TTX". The correct answer is XX but DIFFERENCE will give TXXXXXXXXXXXX.) But, this cannot occur. The argument is as follows. Since /BIG/ must be larger than /SMALL/, it must start with at least the same number of T's. But, if it had exactly the same number of T's as /SMALL/, it cannot have had fewer X's. Thus we would not have performed line 2Ø. So, since we did perform line 2Ø, /BIG/ must have had more T's than /SMALL/.

Finally, we must be sure that SUBTRACT outputs in standard form, since we decided that all of our compact number procedures should deal with numbers in standard form. The output of DIFFERENCE will always have the T's to the left of the X's so the ordering requirement is satisfied. What isn't obvious is that the answer will always have fewer than ten X's. The original /BIG/ and /SMALL/ each have fewer than ten X's (since they are in standard form) so, if line 2Ø isn't used, we're all right. (The difference

between two numbers both less than ten is less than ten.) But, if line 2Ø is used, then /BIG/ will have at least ten X's, often more. How can we be sure that the answer will have fewer than ten? If we were to take away the number of X's that /BIG/ had originally, then the answer would have exactly t n X's. If we take more than that, the answer will have fewer than ten X's. But the only time line 2Ø happens is when the number of X's in /SMALL/ (that is, the number we're subtracting) *is* larger than the original number of X's in /BIG/. So, the answer will have fewer than ten X's.

Choose some representative cases and test SUBTRACT. Notice that it will do *something*, if the inputs are in the wrong order, but the result will be incorrect.

The old MARK-SUBTRACT program that we discussed previously can be made to work with compact numbers by making a very minor change. Line 4Ø was

```
4Ø MAKE
     NAME:  "COUNTER"
     THING:  WORD OF /COUNTER/ AND "X"
```

We'll change that to

```
4Ø MAKE
     NAME:  "COUNTER"
     THING:  ADD OF /COUNTER/ AND "X"
```

This makes no difference as far as mark numbers are concerned since ADD and WORD are the same procedure for them. With compact numbers, however, ADD does more than just WORD the two numbers together.

This SUBTRACT program isn't as "good" as the one we just wrote specially for compact numbers, since it is slower in getting an answer, particularly if the answer is large. It is an interesting

program though, because it depends very little on what method we use for writing numbers. It works equally well for mark numbers, compact numbers, and (as we'll see later) place value numbers. In fact, the reason it is a relatively slow program is that it doesn't take advantage of any of the special shortcuts that may be possible with each particular number notation.

As we find that we want to use larger and larger numbers we'll soon decide that compact numbers, while some help, are not enough. Three lines full of T's is easier to read or write than thirty lines full of X's but both tasks are unpleasant. Fortunately, we can improve the situation somewhat. We can introduce a new symbol, say H, and let each H stand for ten X's. Then we'll have to redefine standard form and rewrite most of our compact number procedures, but when we're finished we will have gained considerably in the size of the numbers we can deal with easily. If we want to work with still larger numbers, we can add yet another symbol standing for ten H's. This process can be repeated until we either have great difficulty remembering all the symbols we introduced or, hopefully, until we can handle any numbers that come up in our work. But, if we need to work with still larger numbers without introducing still more symbols, there are other number representations and some of these will be developed subsequently.

4. More About Mark Numbers

4.1 Multiplication of Mark Numbers

When we wrote the procedure UNCOMPACT, we were actually writing a sort of multiplication procedure. For each T we substituted ten X's. If our original number had only T's in it, the new

uncompacted number would be ten times as long. So, in a way, we have multiplied a mark number by ten. In the process we also changed the marks from T's to X's which, if we had been thinking of compact numbers would have made an important difference. But, with simple mark numbers a mark is a mark (that is, all marks mean the same thing). Nevertheless, let's rewrite UNCOMPACT so that it will take a word of X's (a normal mark number) and substitute ten X's for each X. Since the purpose of this program is different from UNCOMPACT, we'll give it a new name, TIMESTEN.

```
←TO TIMESTEN /N/
>1Ø TEST EMPTYP OF /N/          (We are using the second UNCOMPACT
                                 procedure as our model - the one
                                 that works on disordered compact
                                 numbers)
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (TIMESTEN    (We don't want to check whether
    OF BUTFIRST OF /N/) AND      FIRST OF /N/ is "T". Every mark
    "XXXXXXXXXX"                 in /N/ should have "XXXXXXXXXX"
                                 substituted for it)
>END
←
```

Notice that TIMESTEN is a recursive procedure. The program says that if /N/ isn't /EMPTY/, then TIMESTEN OF /N/ is the same as TIMESTEN OF (BUTFIRST OF /N/) with ten extra X's tacked onto the end (that is, substituted for the one X that BUTFIRST chops off at the beginning).

What we are really after is a general multiplication procedure, not just one that multiplies by ten. We want a procedure that will take two inputs instead of one and will output the product of these inputs. In order to generalize TIMESTEN in this way, we first look carefully at the procedure to see where the fact that we are using <u>ten</u> rather than some other number shows up. The only place in TIMESTEN that this happens is at the end of line 3Ø

where we have "XXXXXXXXXX". If that word were changed, say to "XXXX", TIMESTEN would multiply by four. So, to make TIMESTEN multiply by its second input, we will just put the name of that input in line 3Ø.

```
←TO TIMES /N/ AND /Y/              (Let's change the name of the
                                    procedure since we won't be
                                    multiplying only by ten)
>1Ø TEST EMPTYP /N/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (TIMES OF       (Instead of tacking on ten X's,
    BUTFIRST OF /N/ AND /Y/)        we will use the number given in
    AND /Y/                         /Y/)
>END
←
```

Let's TRACE TIMES and watch it run.

```
←TRACE TIMES
←PRINT TIMES OF "XXX" AND "XXXX"
TIMES OF "XXX" AND "XXXX"
  TIMES OF "XX" AND "XXXX"
    TIMES OF "X" AND "XXXX"
      TIMES OF "" AND "XXXX"
      TIMES OUTPUTS ""
    TIMES OUTPUTS "XXXX"
  TIMES OUTPUTS "XXXXXXXX"
TIMES OUTPUTS "XXXXXXXXXXXX"
XXXXXXXXXXXX
←
```

In order to calculate TIMES OF "XXX" AND "XXXX" the computer had to first find TIMES OF "XX" AND "XXXX" and TIMES OF "X" AND "XXXX" and TIMES OF "" AND "XXXX". So the procedure TIMES is actually called four times. Now suppose we try TIMES OF "XXXX" AND "XXX". The answer will be the same as before, XXXXXXXXXXXX, (unless there is a bug in our program) but the process of getting that answer will be different.

```
←PRINT TIMES OF "XXXX" AND "XXX"
TIMES OF "XXXX" AND "XXX"
  TIMES OF "XXX" AND "XXX"
    TIMES OF "XX" AND "XXX"
      TIMES OF "X" AND "XXX"
        TIMES OF "" AND "XXX"
        TIMES OUTPUTS ""
      TIMES OUTPUTS "XXX"
    TIMES OUTPUTS "XXXXXX"
  TIMES OUTPUTS "XXXXXXXXX"
TIMES OUTPUTS "XXXXXXXXXXXX"
XXXXXXXXXXXX
←
```

In this case the procedure is called five times. If we think about how different the two processes we've just watched are, it may seem surprising that they give the same final result. A very good, mathematical question is "why does this happen?". A good answer to this question, and there are many, would be a way of looking at the TIMES procedure that makes it clear that the order of the inputs doesn't change the final result. Most of the important advances in mathematics have come from discoveries of new ways to look at things which revealed hidden relations.

For the particular question we have here, why TIMES gives the same result even if the two inputs are switched around - we'll look at one possible explanation. First we'll relax slightly the restrictions on how we write mark numbers. We've always written the X's in a horizontal line, like XXXXX, but now let's permit ourselves to write in any patterns we please, for example,

```
X   X          X          X
  X      or    X    or    X     X.  Each of these
X   X          X          X X
               X
```

represents the same number. Now, if we go back to the problem of TIMES "XXX" AND "XXXX", we can think of substituting "XXXX"

for each X in "XXX". But, let's write it like this

```
X  X  X
X  X  X
X  X  X
X  X  X
```

For each X in "XXX" we've substituted a vertical row of "XXXX". If we follow the same scheme for TIMES OF "XXXX" AND "XXX" and substitute a vertical row of "XXX" for each X in "XXXX" we get

```
X  X  X  X
X  X  X  X
X  X  X  X
```

. But now, if we look at both of these patterns, we can see that they are the same except for position, that is, by a simple half turn of the paper the three by four turns into the four by three, and vice-versa. So naturally they have the same number of X's and so they are the same mark number.

A property of this type of explanation is that it is good only if the reader is convinced by it. There are no objective grounds on which to judge it right or wrong, its value depends on how it is received. For this reason most teachers tend to collect as many explanations as they can so that when one doesn't "click" they can try another.

To go back to the TIMES procedure, notice that in the second half of the TRACE printout (the lines that go TIMES OUTPUTS) the computer is counting up by threes in one case and by fours in the other. (That is, three, six, nine, twelve, and four, eight, twelve.) The programs stopped when they came to twelve, the first number that they both reach. An interesting question is: if two programs (or people) start counting up by different amounts (in the case we did, by three and four), what numbers will both programs reach?

4.2 Counting, Copying, and Compact Multiplication

We wrote TIMES to work with mark numbers. We didn't think about how it would behave with other inputs. In fact, whenever we create a LOGO procedure we decide, at least implicitly, what sort of inputs are legal. The mere fact that the procedure is written in LOGO means that the inputs must be either words or sentences (LOGO words or sentences, of course, not necessarily English ones) and not something like a dog, a pound of butter, or a moment of joy. Often, as in the case of TIMES, we intentionally restrict the legal inputs still more. TIMES may have only mark numbers as inputs, other LOGO things are illegal. The word *domain* is used to describe the collection of legal inputs to a procedure. We would say that the domain of TIMES is pairs of mark numbers (pairs because there must be a first input, /N/, and a second input, /Y/).

Sometimes, when the inputs to a procedure are LOGO things that are not in the domain, the computer will stop and type an error comment; sometimes the computer will "go into an infinite loop" (that is, work and work but never output anything). Neither of these cases is particularly interesting. But sometimes the procedure will finish its work and output something. When this happens, it sometimes turns out that we've found a new use for an old tool as, for example, when we found we could use MARK-DIFFERENCE to get the difference of two *compact* numbers. This new use might be rather similar to the old one (like using scissors as tin snips), or it might be very different (like using a magnifying glass to start a fire). Either way the tool becomes more useful.

This is the case with the procedure TIMES. The first input, /N/, exists simply to have each of its letters replaced by the second

input, /Y/. So, it doesn't make any difference what those letters were originally. The only thing that matters is how many there are. Any word at all, used for the first input, can be replaced by the mark number containing as many X's as the original word had letters. So, in fact, we can extend the domain to allow the first input to be any LOGO word. That seems to give us nothing new or interesting. This appearance is misleading however. We can use this observation to write a short procedure called COUNT which will output the number of letters in its input. For example, COUNT OF "CAT" is "XXX" (three), COUNT OF "NARNIA" is "XXXXXX" (six), and so forth. The program is simple.

```
←TO COUNT /W/
>1Ø OUTPUT TIMES OF /W/ AND "X"
>END
←
```

TIMES just substitutes an X for each letter in /W/ and so outputs the mark number equal to the number of letters in /W/. Before we get back to TIMES, notice that we've discussed COUNT only with word inputs, not sentences. Think about COUNT of a sentence, remembering that FIRST of a sentence is the first word and BUTFIRST of a sentence is all except the first word.

```
←PRINT COUNT OF "APPLES PEARS PLUMS CHERRIES"
XXXX
←PRINT COUNT OF "APPLESPEARSPLUMSCHERRIES"
XXXXXXXXXXXXXXXXXXXXXXXX
```

Varying the second input to TIMES is even more interesting. The second input is copied over as many times as there are marks (or letters) in the first input. Thus, for example,

```
←PRINT TIMES OF "XXX" AND "CAT"
CATCATCAT
←
```

This use of TIMES seems cute but not very useful. In fact, it is a surprisingly powerful technique that we will use later when we write programs whose outputs will be LOGO instructions. (Recall, in the procedure BUTFIRSTTEN, we used a line that contained BUTFIRST OF BUTFIRST OF BUTFIRST OF BUTFIRST OF and so on. Procedures like TIMES can produce this kind of instruction.)

Right now, however, let's look at what happens when the second input is a compact number. The output of the procedure will be the compact number repeated as many times as the first input, a mark number, says. So, TIMES OF "XXXX" AND "TTX" is "TTXTTXTTXTTX". But, this is really the product of the mark number and the compact number, except that it isn't in standard form. Using this observation, we can write a multiplication for two compact numbers by uncompacting one of them, using TIMES to multiply the resultant mark number by the other compact number, and finally standardizing the result.

```
←TO MULTIPLY /CM/ AND /CN/
>1Ø OUTPUT STANDARDIZE OF TIMES OF
    (UNCOMPACT OF /CM/) AND /CN/
>END
←
```

Notice that we are careful to uncompact the first input of TIMES in line 1Ø and not the second. Doing it the other way would give a program that does something quite different from multiplication. This program for multiplying compact numbers is considerably better than the simple-minded one of converting both numbers to marks, multiplying the mark numbers, and then compacting the result. In fact, with a little bit of improving (to make sure that the smaller input to MULTIPLY is the one that gets uncompacted), this would be a very reasonable and practical multiplication

procedure. To achieve the most efficient multiplication possible, we'd have to consider also the problem of multiplying T's together (such as "TTT" times "TTTT"). With extended compact numbers (X's, T's, and H's), the problem is even more acute since "HHHHHH" times "HHHHH" is equal to about one teletype page (fifty lines) full of H's. We won't study the problem here, however, since we're going to solve it in a completely different way in the next chapter.

4.3 Division of Mark Numbers

In the same way that UNCOMPACT leads to mark multiplication, COMPACT leads to mark number division. One way the problem of division can be looked at is to ask how many groups of a given size can we make from the number we have. But this is precisely what COMPACT does for a special case, division by ten. In COMPACT we asked how many groups of ten there are in the input number. The answer was used as the number of T's in the compact representation of the number. Let's look again at COMPACT to see what will be involved in changing it to a general mark division procedure, that is a procedure for dividing a given mark number by *any* other mark number.

```
←TO COMPACT /WORD/
>1Ø TEST EMPTYP OF /WORD/
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (FIRSTTEN OF /WORD/) AND (COMPACT
    OF BUTFIRSTTEN OF /WORD/)
>END
←
```

Remember that FIRSTTEN would output "T" if /WORD/ had at least ten marks and it would output /WORD/ unchanged if /WORD/ had fewer than ten letters. If we're going to make COMPACT divide by